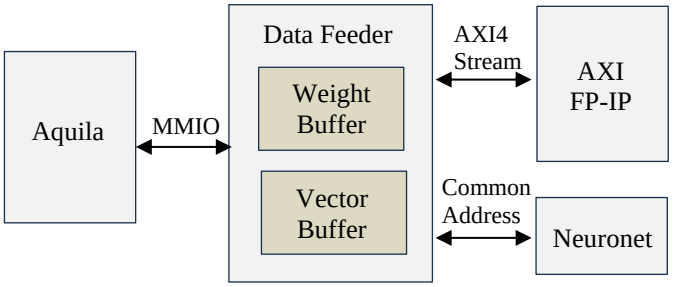
HW#5: Domain-Specific Accelerators

Chu Chih Ling 朱致伶, 110550062

***Abstract*—This report focuses on integrating a domain-specific accelerator (DSA) to Aquila to improve the computing speed of an multi-layer perceptron (MLP) neural network. We propose an effective method to integrate a domain-specific accelerator (DSA) with memory-mapped I/O (MMIO) and AXI-Stream as on-chip bus protocol and analyze the acceleration with different settings. In the experiment, the added vector floating-point HW IP is set to both non-blocking and blocking flow control mode to compare the speed acceleration. Also, the original and optimized string.c is implemented to compare the computation time.**

I. INTRODUCTION

In this report, we first look at how Aquila_SoC module triggers multiple devices by sending master signals to multiple slave signals through I/O device ports of Aquila module. Secondly, we will have detailed description of how the added floating-point IP (FP-IP) get vectors for fused multiply-add operation from memory. Then, introduce the usage of FP-IP with core in both non-blocking and blocking flow control mode. Finally, we compare the time acceleration of different settings, and measure the time spent on data feeding and vector inner-product computations, respectively. The workspace of this assignment adds the OCR in aquila_sw to implement computation. There are three modified modules, soc_top, data_feeder and sram_df, which is different from previous homework workspace. Also, the SD card stores the vectors and weights of multi-layer perceptron (MLP) network and inserts into the socket JD through SD card daughter card on Arty.



II. BUS BEHAVIOR WITH MMIO

The memory-mapped I/O (MMIO) model is one of the common communication interface between processors and device of a system. MMIO allows some memory address to mapped to the internal control registers of the device, including domain-specific accelerator (DSA) used in this assignment. MMIO model makes the processor efficient to write data into addresses to control the device and read data from addresses to retrieve results from the device. The MMIO model is implemented in soc_top module with the bus master and slave component.

A. Bus Master, Slave and Controller

Before introducing the MMIO interface of data feeder, I first introduce the concept of bus master and slave component, which is utilized with the MMIO model. A bus master component initiates read and write requests and bus slave component responds to read/write requests then signal back to the master. In this assignment, there are two pairs of master/slave bus component: (Aquila / Data Feeder), (Data Feeder / FP-IP).

Bus controller is often used when there are multiple bus master and slave components, which is composed of an arbiter and a decoder. The Arbiter determines which master gets to use the bus, and decoder determines which slave should reply to the request. Aquila has both device address decoder and memory arbiter; however, the memory arbiter is left unmodified in this assignment, only device address decoder is used, which is mentioned in the later paragraph.

B. Memory-mapped I/O model

First, I would like to introduce bus signals for MMIO. There are typically seven signals, which is listed below:

- **strobe**: Often combined with the output signals of device address decoder to trigger the device.
- **addr**: Stores the address of output data for slave device.
- **rw**: Determine whether it is a read or write request.
- **byte_enable**: Record respectively whether the bytes in the word are enabled. Since Aquila loads a word at a time, instead of a byte.
- **ready (from device)**: Record whether the bus component is ready to receive the input signals from slave.
- **data in** / **data out (from device)**: Data from and to corresponding slave.

With the knowledge of typical bus signals, the memory-mapped I/O (MMIO) mechanism is explained in this paragraph. Tracing the address output signal from Aquila master can be an entry point of the mechanism. In aquila_top.v, it defines the system memory addresses mapped with tightly-coupled memory (TCM), cached DDRx DRAM, uncached device memory and uncached system device. Since the device of Aquila includes three devices, which are UART board, serial peripheral interconnect (SPI) and domain-specific accelerator (DSA) device, the device memory is segmented to three small pieces in soc_top.v. By checking the dev_addr signal from aquila_top.v, the response slave device, device data output (dev_dout) and device ready (dev_ready) signals are assigned. The corresponding memory of different device declared in soc_top.v is listed below:

- **UART**: 0xC000_0000 – 0xC0FF_FFFF
- **SPI**: 0xC200_0000 – 0xC2FF_FFFF
- **DSA**: 0xC400_0000 – 0xC4FF_FFFF

The communicating process of Aquila and DSA device can be divided into four major steps: 1) Aquila I/O port outputs typical bus master signals listed above. 2) The device address decoder decodes the output address signal to determine which slave device will response to the master request with memory-mapped I/O. In this assignment, data feeder module response to the master request when device address corresponds to the DSA memory space. 3) Data feeder receives the bus signals as slave component, then triggers the floating-point IP. 4) dev_dout and dev_ready are assigned, then send back to Aquila.

III. DATA FEEDER

This section discusses the design of data_feeder module. Data feeder is responsible for the communication between SRAM and floating-point IP. Since the bottleneck of accelerated system is data-feeding of data streams to the AXI FP-IP, data feeder is highly associated to the process of improving the computing speed of multi-layer network (MLP). The process is designed into four steps below: 1) Copy the network neuron and weight value to assigned address in SRAM. 2) Store the value of neurons needed to be computed in specific register, which the value equals for loop iteration times. 3) Trigger the data feeder from neuronet.c, then the data feeder interacts with floating-point IP to implement the inner-product computation in hardware circuit. 4) Assign the computed result to specific pointer in neuronet.c with corresponding address in hardware circuit. 5) Indicate the computed result ready and do the ReLU activation. The detailed implementation is shown below:

A. Variable Declaration

Since neuronet and data_feeder interact with multiple common variables, I declare pointers with specific address in neuronet.c to store states and values during the process.

- **p_dsa_trigger(0xC400000C)**: Trigger the inner-product computation after finish copy value of neurons and weights with memcpy. The value is 0 or 1.
- **p_dsa_count(0xC4000004)**: Stores the count of neurons needed to be computed.
- **p_dsa_ready(0xC4000000)**: Indicate whether the inner-product computation is finished by floating-point IP. The value is 0 or 1.
- **p_dsa_result(0xC4000008)**: Stores the inner-product computed result value.
- **p_vector_sram(0xC4001000)**: Continuous address memory stores the neuron values of multi-layer network.
- **p_weight_sram(0xC4002000)**: Continuous address memory stores the weight values of multi-layer network.

B. Data Feeder & Neuronet

The process of improving the computing speed of multi-layer network (MLP) is implemented with circuiting the neuron network computation. For each computation iteration, the network neurons and weights are copied to p_vector_sram and p_weight_sram with memcpy from string.c. Later, p_dsa_trigger is assigned true in neuronet.c, triggers the inner-product computation to start in hardware circuit in data_feeder.v.

In data_feeder.v, the S_DEVICE_addr_i determines whether S_DEVICE_data_o is a data feeder variable or copied value from SRAM with bit[11:10]. The following paragraph will introduce the two cases: data output is a data feeder variable, or a copied value from SRAM, respectively.

For cases with S_DEVICE_data_o being a data feeder variable, it has to determine which variable should response to the input signals by S_DEVICE_addr_i. There are four possible data feeder response variables. The two of the variables, p_dsa_trigger and p_dsa_ready, indicate status of data feeder. The other two response data feeder variables, p_dsa_result and p_dsa_count, store the current computed result of multi-layer network and the iterations of inner-product needed to be implemented. df_count is a variable storing the current iteration has been implemented. Below is part of the corresponding code in data_feeder.v:

```
1 | assign S_DEVICE_data_o = (!S_DEVICE_addr_i[11:10]) ? df_o : bram_o;
2 | assign df_ready_sel = (S_DEVICE_addr_i[1:0] == 2'b00);
3 | assign df_count_sel = (S_DEVICE_addr_i[1:0] == 2'b01);
4 | assign df_trigger_sel = (S_DEVICE_addr_i[1:0] == 2'b11);
5 | assign df_o = df_ready_sel ? data_ready : df_count_sel ?
data_feeder_counter : df_trigger_sel ? dsa_trigger : dsa_result;
```

When df_count is smaller than the value of p_dsa_count, in other words, p_dsa_trigger is true and p_dsa_ready is false, the data_feeder first loads inner-product vectors and weights from SRAM, stores the value in a_data, b_data, c_data registers, waiting to be transfer to floating-point IP. Also, the current computed result is loaded from floating-point IP and stores in r_data register. df_count increases by 1 after finishing each iteration. When df_count equals to the value of p_dsa_count, which means the inner-product computation of per neuron is done, p_dsa_trigger is set to false and p_dsa_ready is set to true. Since the computation is finished, we are able to load the inner product result from p_dsa_result in neuronet.c, then do the ReLU activation. Below is main part of the modified code, which is the key hotspot neuranet_eval function in neuronet.c:

```
1 |     p_weight = nn->forward_weights[neuron_idx];
2 |     inner_product = 0.0;
3 |     p_neuron = nn->previous_neurons[neuron_idx];
4 |     memcpy((void *)p_vector_sram, (void *)p_neuron, sizeof(float) *
      (nn->n_neurons[layer_idx-1]));
5 |     memcpy((void *)p_weight_sram, (void *)p_weight, sizeof(float) *
      (nn->n_neurons[layer_idx-1]));
6 |     *p_dsa_cnt = nn->n_neurons[layer_idx-1];
7 |     *p_dsa_trigger = 1;
8 |     while(!(*p_dsa_ready));
9 |     *p_dsa_ready = 0;
10 |    inner_product = *p_dsa_result;
11 |    inner_product += *(p_weight);
12 |    nn->neurons[neuron_idx] = relu(inner_product);
```

C. Data Feeder & Floating-Point IP

As for cases with S_DEVICE_data_o being a copied value variable from SRAM, it must determine whether to load the neuron vector or weight by the [11:10] of S_DEVICE_addr_i. Sram_df module is a data cache with 1024 entries and stores a 32-bit word per entry. There are two sram_df modules in data_feeder.v, which is modified from the SRAM module with dual port. One loads neuron vectors, the other loads neuron

weight. Since there is only one device address input signal at a time, sram_df has no dual port design. However, a 32-bit word is loaded per device address input signal, instead of a byte. Therefore, sram_df module has the byte enable mechanism, which is different from simple SRAM module.

Moreover, data_feeder can load and store neuron's value and weight simultaneously. With two SRAM buffers, data _feeder functions as ping pong buffer, which has less latency when interchange the load/store of neuron's value and weight.

Finally, the loaded neuron's attributes are stored in a_data and b_data registers. c_data stores the current computed result, which is updated each iteration. The three above registers send signals to floating-point IP, then receive the output computed value r_data from floating-point IP.

IV. FLOATING POINT IP

Floating-point IP only supports AXI4-Stream Bus protocol. There are four AXI4-Stream channels used in this designed architecture. A data channel always consists of TVALID and TDATA, plus several optional ports and fields, such as TREADY, TLAST and TUSER. TVALID and TREADY perform a handshake to transfer a message, where the payload is TDATA. TVALID is driven by the source (master) side of the channel and TREADY is driven by the receiver (slave). TVALID indicates that the value in the payload fields is valid. TREADY indicates that the slave is ready to receive data. When both TVALID and TREADY are true in a cycle, a transfer occurs. The two flow control modes, non-blocking and blocking, are both implemented during the assignment.

A. Non-Blocking Mode

Non-blocking mode means that lack of data on one input channel does not block the execution of an operation if data is received on another input channel. Full flow control of AXI4-Stream is not always required, which the AXI4-Stream channels do not have TREADY, that is they do not support back pressure. Only TVALID and TDATA are used in each channel. When all of the present four input channels receive an active TVALID, in other words, the inner-product computation is validated, operation occurs on every enabled clock cycle and data is presented on the output channel, which is r_data payload fields regardless of TVALID.

When implementing non-blocking mode, the latency is set to 0. The valid channel of a_data, b_data and c_data are always true in the implementation. Theoretically, non-blocking mode has better performance than blocking mode, since there is no waiting time between channels. The experiment results are shown in the experiment and result section.

B. Blocking Mode

Blocking mode means that operation execution does not occur until fresh data is available on all input channels. Data loss is prevented by the presence of back pressure, which is TREADY signal, so that data is only propagated when the downstream datapath is ready.

The main difference between non-blocking and blocking mode is the TREADY signal. If the output is prevented from off-loading data for the low TREADY signal, then data accumulates in the output buffer internal to the core. When this output buffer is nearly full, the core stops further operations. This prevents the input buffers from off-loading data for new operations, so the input buffers fill as new data is input. When the input buffers fill, their respective TREADYs are deasserted to prevent further input.

When implementing blocking mode, the max latency is set to 3, which has the similar efficiency as non-blocking mode with latency set to 0.

V. EXPERIMENT AND RESULT

The max latency of non-blocking and blocking FP-IP is respectively set to 0 and 3. From Table 1, it shows that circuit the inner for loop of neuronet_eval function in neuronet.c accelerates the computation speed around 3.16 times compared with the original one. Non-blocking flow control with 0 latency and blocking flow control with max latency equals 3 have similar effects. Updating the string.c to optimized version on E3 platform accelerates the computing speed approximately 6.87 times compared with the original ones, which is a huge progress.

TABLE I. TIME MEASURED IN HAND-WRITTEN DIGITS RECOGNITION TEST

	Time Measured	***Accelerated Ratio***	***Accuracy***
Original	22132 msec	1	85 %
Non-blocking FP-IP w/ original string.c	7000 msec	3.16	
Non-blocking FP-IP w/ optimized string.c	3217 msec	6.87	
Blocking FP-IP w/ origianl string.c	6852 msec	3.23	

Table 2 lists the measured clock cycle for OCR program to implement the data feeding and computation. The clock cycle of data feeding is captured when program counter value is in the range of memcpy program counter. As for the computation clock cycle, it calculates the clock cycles when p_dsa_trigger is true and p_dsa_ready is false.

TABLE II. CYCLE MEASURED IN DATA-FEEDING / COMPUTATION

	Total	***Data Feeding***	***Computation (in Circuit)***
Original	1,144,550,757	925,731,317	0
Non-blocking FP-IP w/ original string.c	514,670,954	291,391,147	3,811,200
Non-blocking FP-IP w/ optimized string.c	355,288,266	129,957,636	3,811,200
Blocking FP-IP w/ origianl string.c	330,867,909	120,823,251	3,810,290

VI. CONCLUSION

It took a long time for me to find starting point of improving the computing speed of multi-layer perceptron network integrated with domain-specific acceleration device. While trying to figure out the entry point, I got to know the

architecture of the whole Aquila better, since previous homework only need to get familiar with parts of Aquila code. During this assignment, I have deeper understanding of SPI, AXI4-Lite and AXI4-Stream bus protocols, memory controller interface generator and so on.

After finishing the courses and all of the five assignments, I have the confidence to claim that I got common knowledge of how a microprocessor works and have the ability to implement simple optimization, such as decrease the cache miss rate and accelerate the computing speed with DSA device. Although it always takes me tremendous time to finish the assignment, the course really benefits me a lot. Such a pleasure completing the microprocessor systems course successfully!